A General Framework for Constructive Meta-heuristics

Randall, Marcus

Published in:
Operations Research/Management Science at Work

DOI:
10.1007%2F978-1-4615-0819-9_7

A General Framework for Constructive Meta-heuristics

Marcus Randall
School of Information Technology
Bond University, QLD 4229

June 14, 2001

Abstract

Meta-heuristic search algorithms, by their very nature, are applicable across a range of optimisation problems. In practice however, meta-heuristics have been tailored to solve particular problems. Recent work by Randall and Abramson [17] has successfully shown that iterative meta-heuristics, such as simulated annealing and tabu search, can be successfully generalised to solve a range of problems without modification though the use of a uniform modelling language. Constructive meta-heuristics, such as ant colony optimisation and geralised random adaptive search procedures, pose more substantial problems to achieve this same level of generalistaion. This paper investigates the issues involved and suggests some measures by which generalisation could be achieved.

Keywords: Constructive meta-heuristics, Ant colony optimisation, Greedy randomised search procedures.

1 Introduction

According to Abramson and Randall [1], p.3:

> "For many years, the holy grail of operations research has been an efficient integer solver which can be applied to a wide range of problems with little modification. In this model, a user specifies a problem using some mathematical notation which is processed directly by the solver."

Such systems can take an arbitrarily specified optimisation problem and solve for an optimal/near optimal solution without the need for costly code modifications. In terms of techniques such as the simplex method and branch and bound, there are well established commercial products. However, for integer optimisation problems (notably combinatorial optimisation problems

(COPs)), these systems are inadequate for solving medium to large problems [14].

Recent work by Randall and Abramson [17] has shown that iterative meta-heuristic search techniques share a number of common features such that a generic solver system can be developed. It was found that an algebraic language based on linked lists closely modeled COPs as they are problems in which different arrangements of discrete objects must be explored. The solver system implemented by Randall and Abramson [17] consists of core meta-heuristic engines as well as common functions to handle constraints, evaluate cost functions and apply common neighbourhood transition operators. The implementation showed that the system could produce competitive solutions to difficult COPs in reasonable computational time. This system has the additional advantage that only a high level description (based on the well-known GAMS) of the problem needs to be provided. Normally, solving optimisation problems requires substantial development time in order to produce a competitive heuristic/meta-heuristic program.

Iterative techniques such as simulated annealing (SA) and tabu search (TS) share many similarities as they are based on neighbourhood search techniques. The question becomes whether constructive techniques can be generalised in a similar way. Unlike an iterative technique that continually refines a solution by using a defined set of local transition operators, a constructive technique gradually builds a solution by adding elemental components.

An integrated system offering a range of meta-heuristic search engines could be used as an alternative to products like ILOG's CPLEX and IBM's OSL, with the advantage that relatively large combinatorial problems could be solved efficiently. This paper will outline some strategies to help generalise constructive techniques such as ant colony optimisation (ACO) and greedy randomised adaptive search procedures (GRASP) with particular reference to COPs. As such, it may be considered as a first step toward the aforementioned goal. This paper is organised as follows. Section 2 describes the two most common constructive meta-heuristics while Section 3 discusses how generalisation can be achieved across these two techniques. Conclusions are drawn in Section 4.

2 Constructive Meta-heuristics

Iterative meta-heuristics such as SA, greedy/hill climbing search and TS are techniques that transform a feasible solution into another feasible solution using a neighbourhood transition operator. This process is repeated over a number of iterations. However, constructive meta-heuristics start from an empty solution and gradually build a complete solution. Due to the standard local search neighbourhoods available and the fact that feasibility

can be preserved throughout the search process, iterative meta-heuristics are relatively easy to generalise. Constructive meta-heuristics, on the other hand, cannot make these claims.

The most widely used constructive meta-heuristics are ACO and GRASP. Both techniques are described in general terms so that generalisation can be discussed in subsequent sections.

2.1 ACO

ACO is an umbrella term encompassing a number of techniques including Ant System (AS) [5], $\mathcal{MAX} - \mathcal{MIN}$ Ant System [13] and Ant Colony System (ACS) [4]. Here ACS is examined as a representative ACO meta-heuristic.

ACS is a population based technique that is designed to simulate the ability of ant colonies to determine shortest paths to food. Although individual ants possess few capabilities, their operation as a colony is capable of complex behaviour.

Within each step of an ACO process, each ant adds an *element* to its solution until some *termination condition* is satisfied. In terms of the TSP [10], the QAP [2] and the job sequencing problem (JSP) [8], "element" and "condition" are interpreted according to Table 1.

Table 1: "Elements" and "condition" for the TSP, QAP, JSP.

Problem	*Element*	*Condition*
TSP	an edge	until a Hamiltonian tour is constructed
QAP	a facility	until all the facilities have been placed
JSP	a job	until all jobs have been assigned an order

Using this general terminology, the basic operations of ACS can be described thus. In discrete time steps, allow each ant to add one element to its solution until the termination condition is satisfied. After adding an element, the amount of *pheromone* on that element is modified. Pheromone is a scalar quantity that allows ants to communicate with one another about the utility of an element. The accumulated strength of pheromone on element i is denoted by $\tau(i)$.

At the commencement of each time step, Equation 1 is used to select the next element s to add to the solution. The elements that are still legal to add to the solution (i.e. those that won't break constraints) by ant k at step r are indexed by $J_k(r)$. $\eta(s)$ is the measure of how good element s would be for the solution, in other words an incremental cost measure (a general calculation strategy for which is given in Section 3.3). In terms of the TSP, this would correspond to the distance between two cities.

Equation 1(a) is a highly greedy selection technique favouring the best combination of pheromone level and cost. In order to ensure that other elements are incorporated into an ant's solution, Equation 2 selects elements probabilistically. This is essential to avoid premature convergence to a small group of elite solutions. The randomness of the selection process is governed by a uniformly distributed number q and a parameter q_0.

$$s = \begin{cases} \arg\max_{s \in J_k(r)} \{\tau(s)[\eta(s)]^\beta\} & \text{if } q \leq q_0 \quad \text{(a)} \\ \text{Equation 2} & \text{otherwise} \end{cases} \tag{1}$$

$$p_k(r,s) = \begin{cases} \frac{\tau(s)[\eta(s)]^\beta}{\sum_{u \in J_k(r)} \tau(u)[\eta(u)]^\beta} & \text{if } s \in J_k(r) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For minimisation (maximisation) problems, the parameter β is negative (positive) so that smaller (larger) costs of each s are favoured. The pheromone level of the selected element is updated according to the local updating rule 3.

$$\tau(s) \leftarrow (1-\rho) \cdot \tau(s) + \rho \cdot \tau_0 \tag{3}$$

Where:

ρ is the local pheromone decay parameter, $0 < \rho < 1$.

τ_0 is the initial amount of pheromone deposited on each of the elements.

Upon conclusion of an iteration (i.e. all ants have constructed a feasible solution), elements that compose the best solution are rewarded with an increase in their pheromone level. Mathematically, Equation 4 relates the new pheromone level to the old.

$$\tau(s) \leftarrow (1-\gamma) \cdot \tau(s) + \gamma \cdot \Delta\tau(s) \tag{4}$$

Where:

$\Delta\tau(s)$ is used to reinforce the pheromone level of the elements that compose the best solution found (see Equation 5). L is the cost of the best solution[1] and Q is a constant that is usually set to 100 [5].

γ is the global pheromone decay parameter, $0 < \gamma < 1$ and

$$\Delta\tau(s) = \begin{cases} \frac{Q}{L} & \text{if } s \in \text{ solution by the best ant} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Figure 1 shows the general algorithm for ACS.

[1]The best solution can either be defined as the *global-best* (the best solution found in all iterations) or the *iteration-best* (the best solution found in the current iteration).

Figure 1: The pseudocode for ACS.

```
Read Problem Model;
Get Parameters(num_ants, β, γ, Q, L, ρ, q0, τ0, termination_condition);
τi = τ0 (1 ≤ i ≤ |τ|);
While(termination_condition not met)
  % iteration level
  S = set of randomly ordered feasible elements;
  Xk1 = Sk (1 ≤ k ≤ num_ants);
  While(each solution is not complete)
    For(k = 1 ... num_ants)
      % step level
      Compute η(s) ∀s, s ∈ Jk(r);
      Xkj =Apply Equation 1;
      Compute C(Xk);
      τ =Apply Equation 3;
    End For;
  End While;
  best_ant =Determine the best solution cost from C(Xk)
  ∀k, 1 ≤ k ≤ num_ants;
  τ =Apply Equation 4;
  If((minimisation problem And C(X_best_ant)< best_cost)
  Or (maximisation problem And C(X_best_ant) > best_cost)
  Or (this is the first iteration))
    best_cost = C(X_best_ant);
  End If;
End While;
Output best_cost;
End.
```

Where:

X_{ij} is j^{th} solution element for ant i.

$C(k)$ is the cost of the solution of ant k.

2.2 GRASP

GRASP is a search technique that consists of two distinct phases per iteration, namely a *construction* and *local search* phase [7, 8, 11]. The first phase builds a new feasible solution from a list of elements, one element at a time. The list itself is produced by ordering the elements with respect to a greedy function. This list is similar to the $J_k(r)$ list in ACS except it is ordered by cost. A restricted candidate list (RCL) is then produced

from the top $y\%$ of the candidate list, where y is a parameter of the process. Each element on the RCL has an equal probability of incorporation into the solution. In many instances, this solution will not correspond to a local optimum and can hence be improved. This is achieved by applying a local search procedure (such as hill climbing, SA and TS) to the solution as the second phase. Hence, GRASP may be considered a hybrid constructive/iterative technique. However, the attributes that distinguish GRASP from other meta-heuristics pertain to its construction phase. Figure 2 gives the general pseudocode for GRASP.

Figure 2: The pseudocode for GRASP.

```
Read Problem Model;
Get Parameters(x, termination_condition);
While(termination_condition not met)
  % iteration level
  X = ∅;
  While(X is not complete)
    % step level
    S = Construct an element list ordered by element cost
    (decreasing order if maximisation, increasing order if
    minimsiation);
    RCL = The top (x/100) * |S| elements of S;
    i = unif_rand(1, |RCL|);
    X = X ∪ RCL_i;
  End While;
  X =Execute local search(X);
  Compute C(X);
  If((minimisation problem And C(X)< best_cost) Or (maximisation
  problem And C(X) > best_cost) Or (this is the first iteration))
    best_cost = C(X);
  End If;
End While;
Output best_cost;
End.
```

3 Generalisations

In order to build a constructive meta-heuristic system capable of solving arbitrary COPs, three aspects need to be considered. These are a) *Modelling languages* - how an optimisation problem can be specified in a uniform way; b) *Constraint processing* - the way constraints are handled in constructive

search and c) *Solution augmentation* - the different ways in which the utility/goodness of an element can be determined.

3.1 Modelling Languages

In order to generalise any optimisation technique, there must be some way of specifying problems in a uniform way to be processed by a solver. Presented herein are two appropriate modelling systems for constructive metaheuristics. The first is based on a directed graph approach while the second uses linked lists.

3.1.1 Graph Based System

Gutjahr [9] describes a general representation called *Graph Based Ant System* for COPs. This system uses a directed graph $\mathcal{C} = (\mathcal{V}, \mathcal{A}, \Phi)$ where $\mathcal{C}$ is the construction graph, $\mathcal{V}$ is the set of nodes (the possible values or elements) and $\mathcal{A}$ is the set of arcs. The central idea is that an ant begins from a unique start node in the graph and walks to the end node. The path taken by the ant defines its solution. Φ is a function that defines the set of feasible walks through the construction graph. However, Φ is very problem specific and may be difficult to define for particular problems. However, $\mathcal{V}$ and $\mathcal{A}$ may be simply specified using either an adjacency matrix or list.

The generalised assignment problem (GAP) will be used to demonstrate the application of the construction graph technique. The aim of the GAP is to assign a number of jobs to a set of agents such that the cost of the assignment is minimised and the capacity of each agent is not violated. This is expressed in Equations 6-9.

$$Minimise \quad \sum_{i=1}^{N} \sum_{j=1}^{M} c_{ij} x_{ij} \tag{6}$$

s.t.

$$\sum_{i=1}^{N} a_{ij} x_{ij} \leq b_j \quad \forall j \quad 1 \leq j \leq M \tag{7}$$

$$\sum_{j=1}^{M} x_{ij} = 1 \quad \forall i \quad 1 \leq i \leq N \tag{8}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \quad 1 \leq i \leq N \quad 1 \leq j \leq M \tag{9}$$

Where:

x_{ij} is 1 if job i is assigned to agent j, 0 otherwise.

c_{ij} is the cost of assigning job i to agent j.

a_{ij} is the resource required by agent j to perform job i.

M is the number of agents.

N is the number of jobs.

There are many ways that the construction graph can be drawn from the one formulation. Figure 3 shows two different construction graphs for a small GAP in which $N = 3$ and $M = 2$. Each one will effect each meta-heuristic's performance differently. For instance, the construction graph of Figure 3(a) forces the search technique to first assign the jobs to agent 1 and then move to agent 2. In this case, the jobs of agent 1 would have to be given biased probability weightings so that the search technique does not move too quickly from agent 1 without the chance to go back. This could lead to a situation in which agent 1 is not assigned enough jobs, potentially leading to poor quality solutions. The graph of Figure 3(b) does not have this restriction. The former graph (a) allows fewer choices than (b) for every node (giving some efficiency gains), but is more restrictive. The number of arcs in the graph is $O(p^2)$ where $p = |\mathcal{V}|$.

GRASP can also use this system in the following way. Assume that a solution to an optimisation problem can be expressed as $S = \{s_0, s_1, s_2, \ldots, s_t\}$ where $t = |S|$. Given that s_a $(a < t)$ is the most recently added element, the arcs leading from s_a can be arranged in decreasing order of cost to be considered as s_{a+1}. An RCL is then used to retain the top $y\%$ and using an equal probability for each candidate, s_{a+1} can be calculated and added to the solution.

3.1.2 Linked List Modelling

A representation technique based on dynamic lists has been used for iterative meta-heuristics by Randall [14] and Randall and Abramson [17]. The technique can best be illustrated by an example based on the GAP, as shown in Figure 4. In a list representation, solutions can be represented using a double nested list of agents, each of which contains a list of associated jobs. Thus, in the example, jobs 1, 3, 5 and 2 are assigned to agent 1.

In linked list notation, the objective of the GAP can be expressed as:

$$Minimise \sum_{i=1}^{M} \sum_{j=1}^{|x(i)|} C(i, x(i, j)) \tag{10}$$

Where:

x is the solution list. $x(i, j)$ is the j^{th} job performed by agent i.

C is the cost matrix. $C(i, j)$ is the cost of assigning job j to agent i.

M is the number of agents.

Figure 3: Two possible construction graphs for a small GAP. (r, s) denotes that job s is assigned to job r.

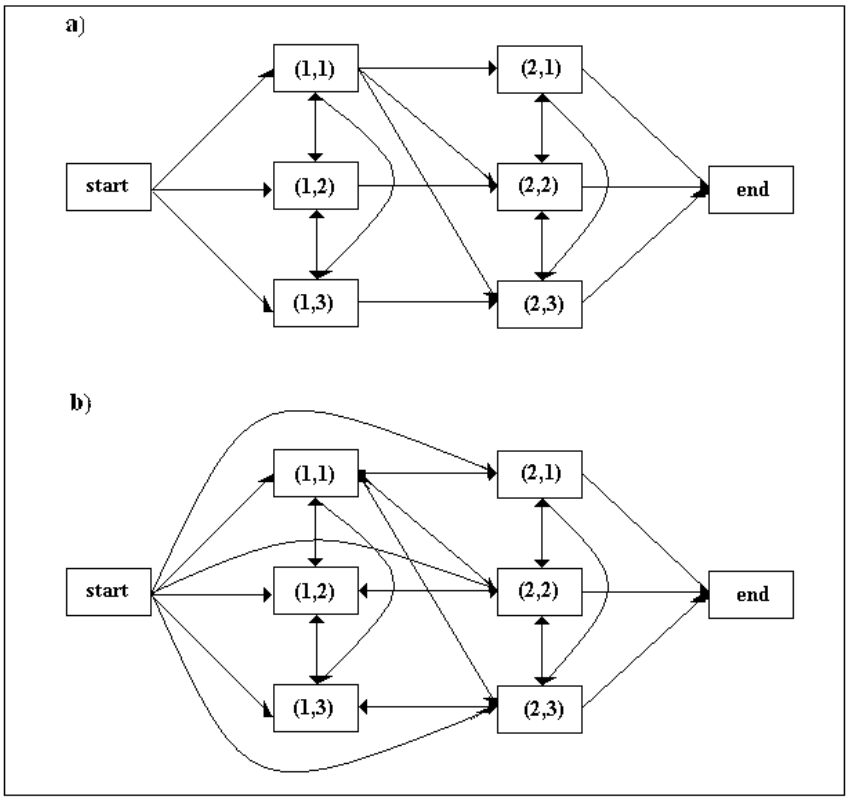
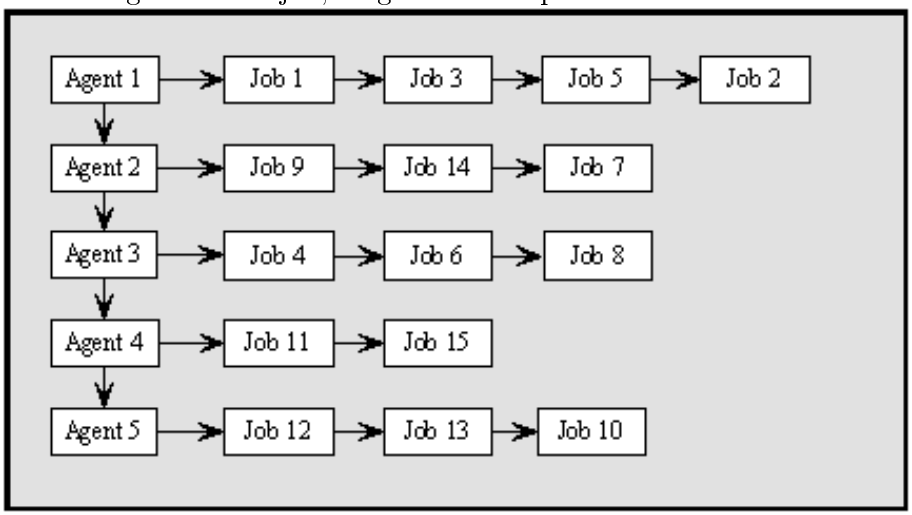


Figure 4: A 5 job, 4 agent GAP represented as a linked list.



Whilst this list notation appears similar to conventional vector form used in standard linear programmes, each sub-list contains a variable number of elements. Thus, the second summation sign in 10 requires a bound which varies depending on the length of each sub-list (i.e. $|x(i)|$) and changes according to the current solution state. Similarly, the constraints concerning the capacity of each agent are formed across list space. In this example, Equations 12 - 15 are the *list constraints* while Equation 11 represents the *problem constraints*.

$$\sum_{j=1}^{|x(i)|} a(i, x(i, j)) \leq b(i) \qquad \forall i \quad 1 \leq i \leq M \tag{11}$$

$$|x| = M \tag{12}$$

$$1 \leq x(i, j) \leq N \qquad \forall i, j \quad 1 \leq i \leq M \quad 1 \leq j \leq |x(i)| \tag{13}$$

$$min\_count(x) = 1 \tag{14}$$

$$max\_count(x) = 1 \tag{15}$$

Where:

a is the resource matrix. $a(i, j)$ is the resource required by agent i to perform job j.

b is the capacity vector. $b(i)$ is the capacity of agent i.

N is the number of jobs.

Equation 12 ensures that the number of agents remains static throughout the search process while Constraint 13 defines the range of element values between 1 and N. The function *min_count* and *max_count* ensure that each value (as specified in Constraint 13) appears at least and at most once in the list structure (i.e. every job is represented once).

The initial solution generation technique described by Randall [14] can be used for arbitrary COPs and consists of two phases. In the first phase, a general constructive heuristic is used to place elements throughout the list structure while the second phase applies a meta-heuristic to attain problem feasibility. A modified form of the first phase can be used with a constructive meta-heuristic in order to solve arbitrary COPs expressed in terms of linked lists. The first stage consists of selecting a set of elements to place on the list and then distributing these elements throughout the list structure. This is presented in Figure 5.

Rather than just taking the $\texttt{i}^{th}$ element of `elements_to_place`, the constructive meta-heuristic can be used to generate the element from the set `elements_to_place`. This is expressed in the pseudocode of Figure 6.

Figure 5: The initial solution generation technique for list models taken from Randall [14].

```
/* Create the vector of element values to place in the solution
list */
Case ((min_count(x)=1) AND (max_count(x)=1))
      For i=list_value_lowerbound, list_value_upperbound
            elements_to_place(i-list_value_lowerbound + 1)=i;
      End For;
Case (max_count(x)=1)
      j=list_value_lowerbound;
      For i=list_value_lowerbound, list_value_upperbound
            If (randomly_select()=true)
                  elements_to_place(j -list_value_lowerbound + 1)=i;
                  j=j+1;
            End If;
      End For;
Case ((min_count(x)=1) OR (unspecified))
      For i=list_value_lowerbound, list_value_upperbound
            elements_to_place(i - list_value_lowerbound + 1)=i;
      End For;
      j=list_value_lowerbound;
      For i=list_value_lowerbound, list_value_upperbound
            If (randomly_select()=true)
                  elements_to_place(j -list_value_lowerbound + 1)=i;
                  j=j+1;
            End If;
      End For;
/* Place the elements */
Randomly rearrange the elements contained in the elements_to_place
vector;
For i=1,length(elements_to_place)
     sublist=succ(x,sublist);
     position=|x(sublist)|;
     While (x(sublist,position) does not violate the list
     length bounds)
          sublist=succ(x,sublist);
          position=|x(sublist)|;
          If (each sublist has been tried without success) exit and
report failure;
     End While;
     x(sublist,position)=elements_to_place(i);
End For;
End.
```

Where:

x is the solution list.
elements_to_place is a vector containing the element values to place on the solution list.
randomly_select() is a function that generates a random state from the set {true,false}.
length(i) returns the length of vector i.
succ(x,i) returns i+1 unless i =|x| at which i=1.

3.2 Constraint Processing

As constructive meta-heuristics, by and large, do not operate on complete solutions, it may be difficult to evaluate some constraints. In general, there are three ways of dealing with constraints within a meta-heuristic framework:

- *Feasibility Maintenance.* This method ensures that at each step of the algorithm, the solution (or group of, if the technique is population based) is feasible. The advantage is that at the end of the iteration, a feasible solution will be produced. In addition, feasibility

Figure 6: The pseudocode for integrating the list initial solution generation technique and a constructive meta-heuristic.

```
elements_to_place = Determine the set of elements on the solution list
based on the count constraints;
While(iteration not finished);
   (sublist,position)=Determine a list position that does not violate
   the list constraints;
   element=Select an element from elements_to_place based on the
   specific meta-heuristic rules;
   x(sublist,position)=element;
End While;
End.
```

maintenance is generally less computationally intensive than feasibility restoration [1] (discussed next).

- *Feasibility Restoration.* In this method, an element is added without regard to feasibility. If the solution becomes infeasible as a result, a heuristic is invoked that changes other elements of the solution in order to produce a feasible solution again. The disadvantage is that the feasibility restoration process can be very computationally intensive and problem specific. However, as infeasible space is traversed, solutions that may be unreachable by other means, are potentially explored. It remains an open question as to whether an efficient general purpose feasibility restoration technique can be produced for constructive meta-heuristics.

- *Feasibility Ignorance.* In this method, the feasibility of a particular element (as it is added) is not considered. Whilst this approach is very efficient, many complete solutions will be infeasible (depending on the tightness and the number of constraints).

In this section, ideas for implementing a feasibility maintenance scheme will be described. Generic feasibility restoration is a difficult topic and some preliminary work is being done by Randall [16].

It can be more difficult to apply a feasibility maintenance approach to a constructive heuristic (such as ACO) than an iterative heuristic (such as SA and TS). This is because it may be difficult to know when (at what step) to terminate the construction process. However, termination is dependent on the type of constraints present in the problem model. There are two types of constraints:

1. *Constraints that must be satisfied at each step.* An example of such a constraint is a knapsack capacity constraint of the form $\sum_{i=1}^{N} c_i x_i < b \quad x_i = \{0, 1\}$. Regardless of the number of steps that have elapsed within an iteration, it is always possible to satisfy such a constraint.

2. *Constraints that cannot be satisfied at each step of the algorithm.* Consider the case where a constraint of the form $\sum_{i=1}^{N} c_i x_i > b$ is present in the problem model. Clearly in the beginning stages of the construction, the length of the solution vector is small and it is unlikely that this constraint will be satisfied. Hence, these types of constraints must be treated as special in the design and develoment of constructive solvers. Only once these types of constraints have been satisfied can the algorithm consider the termination of the solution augmentation process. Thus, once all such constraints have been satisfied, the agent (for instance, the ant in the case of ACO) can be terminated. This defines the generic iteration termination condition.

3.3 Solution Augmentation

Unlike iterative techniques that can use a variety of standard local search operators (such as swap and move), constructive heuristics have one broad operator: to add an element to the solution. The way to calculate which element to add depends on element cost, constraints and meta-heuristic specific mechanics. For both ACO and GRASP, this function produces a probability for each element. As each element must be examined, it can be computationally expensive to complete an iteration of the search process. Candidate set strategies are used to limit the set of elements examined at each step. Randall [15] has proposed some generic candidate set strategies.

For constructive meta-heuristics there are three general ways to calculate the utility of an element.

1. *Element Cost.* This approach simply calculates the cost of an element and is suitable for problems such as TSP where the cost function consists of adding a set of separate costs together (Equation 16).

$$\Delta C_e = f(e) \tag{16}$$

Where:

ΔC_e is the incremental cost of adding element e to the solution

$f(e)$ is a problem specific function specifying the incremental cost of element e.

2. *Element Cost + Constraint costs.* In addition to the above point, the cost of an element is offset by the amount it violates or potentially violates the constraints. Using the system outlined in Randall and Abramson [17], this degree can be calculated. Constraint violation can be calculated according to the relational operators present in the constraints. For instance, if the sign of a constraint is $\leq$ and the left-hand side is larger than the right-hand side, the net difference is the amount of constraint violation. This is shown in Equation 17. The constraint violations (Equations 18-22) for the other signs are calculated in a similar matter. The total element cost is given in Equation 23. The operand in this equation is $+$ if the objective function is to be minimised, $-$ for maximisation.

$$(\leq) \quad cv_i = MAX(0, lhs_i - rhs_i) \tag{17}$$

$$(<) \quad cv_i = MAX(0, lhs_i - rhs_i + 1) \tag{18}$$

$$(\geq) \quad cv_i = MAX(0, rhs_i - lhs_i) \tag{19}$$

$$(>) \quad cv_i = MAX(0, rhs_i - lhs_i + 1) \tag{20}$$

$$(=) \quad cv_i = |lhs_i - rhs_i| \tag{21}$$

$$cv_i = \begin{cases} if(rhs_i = lhs_i), 1 \\ 0, otherwise \end{cases} \tag{22}$$

$$\Delta C_e = f(e) \pm \sum_{i=1}^{R} cv_i \tag{23}$$

Where:

R is the number of constraints

cv_i is the constraint violation of constraint i

lhs_i is the evaluation of the left-hand side of constraint i

rhs_i is the evaluation of the right-hand side of constraint i.

3. *Approximation Function.* An approximation function would be used if either it is not possible to calculate the impact of the element without reevaluating the entire solution, or it would be too difficult or computationally intensive to calculate the effect of an individual element. The latter represents the case of the QAP, where calculating the cost of individual elements is $O(n^2)$ and calculating the cost of the entire solution is also $O(n^2)$. In such cases, an approximation function would be a function with a lower worst case complexity that estimates the cost of the element. The actual solution cost can be calculated at the end of the iteration. Equation 24 gives the general form of the approximation function.

$$\Delta C_e = ap(e) \tag{24}$$

Where:

ap is the problem specific approximation function.

For some complex problems that have difficult cost functions and constraints, a mixture of methods 2 and 3 may be appropriate. If the linked list modelling system is used, its GAMS like language could be modified to accommodate the type of incremental cost function required.

Using the above notions, it may be possible to broaden the definition of the term "element". Instead of adding an individual element, a group of elements could be added at each step of the algorithm. A set of possible groups would need to be predefined. An element group consists of one or more individual element that work well together (i.e. have a collective high utility). This approach has been used by Randall, McMahon and Sugden [18] to solve network design and synthesis problems. In this approach, a subset of network routes are developed before the running of the SA meta-heuristic. This subset consists of routes that produce low cost solutions and help to maintain problem feasibility. Using the candidate grouping technique, a constructive meta-heuristic needs fewer steps per iteration in order to construct a solution. This of course reduces computational time.

4 Conclusions

Practitioners and researchers alike build special purpose meta-heuristic codes in order to solve combinatorial optimisation problems. This is time consuming and does not allow for easy comparison between implementations. However, a uniform framework as proposed in this paper can be used in order to build a general solver. Such a solver would allow a model developer to rapidly prototype problems and solve them using relatively new constructive meta-heuristic techniques.

The issues that have been addressed in this paper are: modelling approaches including constructive graphs and linked list modelling; constraint processing and solution augmentation. As shown, these issues can be treated in a general way, whereas most practitioners/researchers treat these issues differently for each problem they solve. As such they are applicable for a general solver as well as tailored solvers. A general constructive meta-heuristic solver needs to be built in order to implement, test and extend the ideas contained herein. An empirical analysis of the performance of such a system also needs to be undertaken. Given the results of Randall and Abramson [17] for general iterative meta-heuristics, it is conceivable that such a system would be competitive with customised meta-heuristic and heuristic programmes.

References

[1] Abramson, D. and Randall, M. (1999) "A Simulated Annealing Code for General Integer Linear Programs", Annals of Operations Research, 86, pp. 3-21.

[2] Burkard, R. (1984) "Quadratic Assignment Problems", European Journal of Operational Research, 15, pp. 283-289.

[3] Corne, D., Dorigo, M. and Glover, F. (1999) New Ideas in Optimisation, The McGraw-Hill Company.

[4] Dorigo, M. and Gambardella, L. (1997) "Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem", IEEE Transactions on Evolutionary Computation, 1, pp. 53-66.

[5] Dorigo, M., Maniezzo, V. and Colorni, A. (1996) "The Ant System: Optimization by a Colony of Cooperating Agents", IEEE Transactions on Systems, Man and Cybernetics - Part B, 26, pp. 29-41.

[6] Feo, T. and Resende, M. (1989) "A Probabilistic Heuristic for Computationally Difficult Set Covering Problems", Operations Research Letters, 8, pp. 67-71.

[7] Feo, T. and Resende, M. (1995) "Greedy Randomized Adaptive Search Procedures", Journal of Global Optimization, 51, pp. 109-133.

[8] Glover, F. and Laguna, M. (1997) Tabu Search, Kluwer Academic Publishers, Boston, 402 pages.

[9] Gutjahr, W. (2000) "A Graph-based Ant System and its Convergence", Future Generation Computer Systems, 16, pp. 873-888.

[10] Lawler, E., Lenstra, J., Rinnoy Kan, A. and Shmoys, D. (1985) The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, Chichester, 465 pages.

[11] Mavridou, T., Pardalos, P., Pitsoulis, L. and Resende, M. (1995) "Parallel Search for Combinatorial Optimization: Genetic Algorithms, Simulated Annealing, Tabu Search and GRASP", Proceedings of Parallel Algorithms for Irregularly Structured Problems, and Springer-Verlag, Lecture Notes in Computer Science, 980, (1995), Eds: Ferreira, A. and Rolim, J., pp. 317-331.

[12] Pardalos, E. and Wolowiczi, H. (1994) Dimacs Series on Discrete Mathematics and Theoretical Computer Science, 16.

[13] Ramalhinho, H. and Serra, D. (1998) "Adaptive Approach Heuristics for the Generalized Assignment Problem", Working Paper Series, No. 288, UPF.

[14] Randall, M. (1999) "A General Modelling Systems and Meta-heuristic based Solver for Combinatorial Optimisation Problems," PhD thesis, Faculty of Environmental Science, Griffith University. Available online at: `http://www.it.bond.edu.au/randall/general.pdf`

[15] Randall, M. (2000) "Candidate Set Strategies for Ant Colony Optimisation", Submitted to IEEE Transactions on Evolutionary Computation.

[16] Randall, M (2000) "The Use of Feasibility Restoration in Iterative Meta-heuristics for Solving Combinatorial Optimisation Problems", Working Paper.

[17] Randall, M. and Abramson, D. (2000) "A General Meta-heuristic Based Solver for Combinatorial Optimisation Problems", Journal of Computational Optimization and Applications (to appear).

[18] Randall, M., McMahon, G. and Sugden, S. (2000) "A Simulated Annealing Approach to Communication Network Design", Journal of Combinatorial Optimization (to appear).